Controlling Multiple Manipulators Using RIPS

Yulun Wang Steve Jordan Amante Mangaser Steve Butner

Center for Robotic Systems in Microelectronics
University of California, Santa Barbara

Abstract

A prototype of the RIPS architecture - Robotic Instruction Processing System - has been developed at the Center for Robotic Systems in Microelectronics, University of California at Santa Barbara. A two-arm robot control experiment is underway in order to characterize the architecture as well as research multi-arm control. This experiment uses two manipulators to cooperatively position an object. The location of the object is specified by the host computer's mouse. Consequently, real-time kinematics and dynamics are necessary. The RIPS architecture is specialized so that it can satisfy these real-time constraints.

This paper discusses the two-arm experimental set-up. A major part of this work is the continued development of a good programming environment for RIPS. We are working with the C++ language and have favorable results in the targetting of this language to the RIPS hardware.

1. Introduction

RIPS, for Robotic Instruction Processing System, is a specialized computer targetted to meet the real-time computational requirements for advanced robot control [1-3]. Although the architecture assumes no manipulator characteristic or control strategies, its unique structure can extract most of the parallelism inherent to robot control problems.

A prototype system has been designed and built. This system is being used by researchers at the Center for Robotic Systems in Microelectronics at the University of California, Santa Barbara, to experiment with robot control algorithms which were previously too computationally intensive for real-time evaluation.

This paper describes our current efforts in a two-arm cooperative manipulation experiment. This experiment will enable us to analyze the performance of RIPS as well as study two-arm control. Since RIPS is a custom system, a major part of this work has been the development of a programming environment which can support robotic research. Section 2 gives a brief description of the RIPS architecture. Section 3 describes the two-arm experiment. Section 4 discusses the programming environment which we are developing, and section 5 offers some concluding remarks.

2. The RIPS Architecture

RIPS is a multiprocessor architecture where a tightly coupled cluster of processors is allocated to each dynamically coupled system. Multiple clusters are used to control multiple manipulators. The system level architecture is shown in Figure 1.

Within a cluster, a private bus controlled by a custom I/O (DMA) handler provides high-speed interprocessor communication. Data transfers between processors of the same cluster require a 3 microsecond set-up time, and 400 nanoseconds per 32-bit word transfer. Therefore, multiple transfers are possible within the servoing update cycle.

Communication between clusters is supported by a standard asynchronous bus (VME), which operates at a slower speed. This bus is used for higher level, and hence slower, communication. For example, this bus can be used to coordinate the motion of multiple mechanisms.

The RIPS architecture uses extensive parallel processing to increase the real-time execution speed of robot control algorithms. Before parallel processing can be applied, however, it is important to realize that parallelism can be exploited at many different levels. For example, partitioning the target problem into multiple sub-problems, and simultaneously executing the sub-problems is considered *job level* parallelism. Whereas, pipelining the instruction execution of a processor is considered *intra-instruction level* parallelism. A detailed understanding of the target problem is essential before parallel processing can be effectively applied.

In the RIPS system, each of the subsystems simultaneously executes a different part of the robot control problem. The general schema is a convenient user interface at the host level, which issues trajectory-level commands to the robotic processor(s). The robotic processor evaluates the inverse kinematic and inverse dynamic equations for a particular manipulator, and uses its I/O handler for interprocessor communication. The servo controllers run independently using (optionally) a higher speed update cycle to servo about trajectory set points.

The robotic processor is a custom processor designed to exploit parallelism in robot kinematics and dynamics. After examining kinematic and dynamic equations, we found that they can be efficiently formulated using three-dimensional vector equations. In fact, any rigid body dynamics problem can be expressed in three-dimensional vector notation. An intuitive reason for this intrinsic structure is that these equations explain the motion of three-dimensional bodies in a three-dimensional space. Consequently quantities like positions, velocities, accelerations, forces, and moments are most conveniently described by 3-D vectors. A detailed explanation of the robotic processor's operation can be found in [3].

A prototype RIPS system has been built and is currently being used for a two-arm manipulation experiment. The current configuration consists of a SUN 3/140 host computer, one robotic processor, one I/O handler, and one servo controller.

3. A Two-Arm Experiment

Advances in multi-arm cooperative manipulation will enhance the capabilitties of robots tremendously. One of the major difficulties which prevents the development of such systems is that the computational requirements quickly become overwhelming, even with moderately complex manipulators performing seemingly simple tasks. We felt that RIPS can offer good experimental data to this area of research.

Our experiment is designed both to provide insight into two-arm control as well as to test the RIPS architecture. The two-arm set-up is shown in Figure 2. Each arm is a five-bar-link direct-drive mechanism, which was designed and built at the CRSM [4]. Three direct-drive motors are used to control each manipulator. Even though these manipulators can move only in a horizontal plane, our control programs assume general 3-D capabilities. This is so that our results can be extrapolated and applied to more complex mechanisms.

Our experiment demonstrates real-time two arm cooperation. The SUN host's mouse generates a trajectory for an object which is held by both arms. Since the motion of the mouse is not preplanned, the inverse kinematics must be computed in real time. If high speed manipulation is required, real-time inverse dynamics is also needed.

3.1 Control Strategy

When two arms cooperatively manipulate an object, the entire system (i.e. the two arms and the object) becomes dynamically coupled. If the inverse dynamics of this system is solved with a single recursive algorithm, the computational burden becomes enormous. Nakamura [5] proposed a dynamic force control method which allows parallel processing to be used. A block diagram of this method controlling two cooperating manipulators is shown in Figure 3. At the beginning of each update cycle, the simple dynamics of the object is calculated to determine the trajectory of each manipulator. This information is fed to the servo blocks. The servoing of each arm, which includes the inverse dynamics and kinematics, is computed in parallel, reducing the computation time considerably.

Our prototype system has only one copy of each processor. As discussed earlier, ideally one cluster of processors is assigned to each manipulator. However, due to limited resources we perform the computations for both manipulators with one set of processors. The high computational capacity of these subsystems allows us to maintain a good update rate (~1 ms) anyways.

3.2 Sensor data acquisition and Filtering

The manipulators' actuators and encoders are accessed though a custom interface card. This card has 6 optical encoder input channels and 6 analog output channels. Multiple cards can be used simultaneously if additional joints are to be controlled.

Accurate position and velocity measurements are mandatory for good trajectory control. The interface card uses standard optical encoder feedback to measure joint position. Joint velocity is measured by using a high frequency clock to compute the time between successive rising edges from one of the two optical encoder channels. Velocity is derived by the servo controller with a simple inversion algorithm. Only one of the two channels is used since optical encoders do not always maintain their two channels 90 degrees out of phase. The direction of rotation is also incorporated into the velocity signal by noting the change in position. A 2 pole low-pass filter was necessary to eliminate high-frequency noise.

Figure 4 shows an example of the resulting velocity signal before and after filtering. The corresponding position is displayed for reference. This graph shows a manipulator joint's response to an input step. Note that we underdamped the system to create a more demonstrative picture.

4. Programming Environment

We have spent a good deal of time developing a programming environment for RIPS. A software support will allow RIPS to be used more effectively in robot control research, such as our two-arm experiment. Since our processing subsystems are custom designed, we have had to develop our software starting from a very low level. Fortunately the host operating system is UNIX, so all of the UNIX programming tools are available. These tools have assisted us tremendously in our development efforts.

A UNIX device driver lets us communicate with the different processing subsystems. A menu-driven monitor program assists the user in issuing commands. Figure 5 gives a snapshot of its format. The left column contains the system level commands. Some of these commands have sub-menus which are displayed in the right column when requested. Figure 5 shows the sub-menu for the robotic processor in the right column. This menu is displayed when the user types the command 6 (for rpmenu). Our eventual goal is to develop an interface which appears identical to the UNIX shell. Our shell, however, will understand the additional hardware resources (i.e. robotic processors, I/O handlers, and servo controllers) and know how to use them. With this, a user can run a program which is automatically executed on the custom hardware.

Good program development support is essential for an experimental system. This begins from the lowest-level translators. A TMS320C25 assembler donated from Texas Instruments was ported from MS-DOS to UNIX so that we can write I/O handler and servo controller programs. We had to write a custom assembler for the robotic processor because of its unique instruction set. This assembler was designed so that the instruction set can be easily modifed. This has turned out to be a very useful; we have already updated the robotic processor's instruction set twice.

4.1 High-Level Language Programming

High-level language support allows other users to easily write programs for the system. We have already developed a number of large assembly language programs and are well aware of the inconveniences of assembly language programming.

We are first targetting a high-level language to the robotic processor since it executes the most difficult algorithms. The I/O handler and servo controller programs should remain fairly simple, hence assembly language programming is not too cumbersome. The I/O handler only executes a fairly simple polling program for interprocessor communication. The servo controller interfaces to the manipulators, and calculates simple control laws. Eventually we will provide high-level support for these subsystems.

We have decided to support C++ as the high-level language [6]. The main advantages of C++ are that it is object-oriented, it supports user-defined types, and it permits function overloading. Instead of starting from scratch, we are retargetting the GNU compiler from the Free Software Foundation [7]. This compiler is distributed freely in source code form, and has already developed a reputation for its high quality code. The GNU optimizing C and C++ compilers share the same retargettable back end, but have different front ends.

The robotic processor has an original architecture which creates some original compiler problems. For example, the same register file is used to store both vector and scalar operands. Therefore, the compiler must be taught the relationship between vector and scalar registers.

We have already successfully retargetted the GNU C compiler to the robotic processor. The major effort involved writing a machine description which explains the processor's architecture to the compiler. This compiler allows us to write C programs which execute on the robotic processor. However, it cannot exploit the processor's vector capabilities. This limitation is partly due to the C language itself. The advantages of C++ should be helpful in remedying this problem. C++ supports the concept of user-defined types, which is explained in the next section.

4.2 C++, the Robotic Processor, and Robot Control

The robotic processor is optimized for operating on 3-dimensional vectors. Since C++ supports user defined types we can exploit this characteristic by defining a new type, VEC3. VEC3 stands for 3-dimensional vector. This new type enables the compiler to recognize the three-fold parallelism inherent in the program, and exploit the underlying hardware. This is best explained with an example.

Computing the Jacobian matrix

The Jacobian matrix relates joint velocities to Cartesian velocities, and is used throughout various robot control techniques. Orin and Schrader developed a very efficient algorithm for computing the Jacobian [8]. The equations which implement the algorithm are:

$$^{N+1}\mathbf{U}_{N+1} = \mathbf{I}$$

$$^{N+1}\mathbf{U}_{i-1} = {}^{N+1}\mathbf{U}_i \, {}^{i-1}\mathbf{U}^T_i \qquad i = (N+1), \ldots 2, 1$$

$$^{N+1}\gamma_{i-1} = {}^{N+1}\mathbf{U}_{i-1}\begin{bmatrix}0\\0\\1\end{bmatrix} \qquad i = 1, 2, \ldots N \quad \text{revolute joint}$$

$$^{N+1}\gamma_{i-1} = 0 \qquad i = 1, 2, \ldots N \quad \text{prismatic joint}$$

$$^{N+1}\mathbf{r}_{N+1} = 0$$

$$^{N+1}\mathbf{r}_{i-1} = {}^{N+1}\mathbf{r}_i - {}^{N+1}\mathbf{U}_i \, {}^{i}\mathbf{p}^*_i$$

$$^{N+1}\beta_i = {}^{N+1}\gamma_i \times (-{}^{N+1}\mathbf{r}_i) \quad i = 1, 2, \ldots N \quad \text{revolute joint}$$

$$^{N+1}\beta_{i-1} = {}^{N+1}\mathbf{U}_{i-1}\begin{bmatrix}0\\0\\1\end{bmatrix} \qquad i = 1, 2, \ldots N \quad \text{prismatic joint}$$

where the **U**s are 3-by-3 rotational matrices, the **I**s are 3-by-3 identity matrices, and the βs, γs, **r**s and $\mathbf{p}^*$s are 3-dimensional vectors. β_i and γ_i form the ith column of the Jacobian. This algorithm starts at the manipulator's end-effector and recursively propagates velocities back towards the base. The resulting Jacobian transforms joint velocities to Cartesian velocities with reference to the manipulator's end effector. Since the Jacobian matrix transforms joint velocities to linear and angular velocities, the algorithm is most efficiently expressed using only 3-D vectors and 3-by-3 rotational matrices.

As mentioned earlier, C++ lets the user define new data types - or *classes* - as well as the operator functions when operating on data of the new types - or *overloading*. The VEC3 data type is instrumental in programming this algorithm so that the robotic processor can exploit the algorithm's inherent parallelism. The three components of a VEC3 object are i, j, and k. These components can be accessed individually as well as collectively. We overloaded the arithmetic operators so that the C++ compiler understands how to manipulate VEC3 objects. Addition, subtraction, multiplication, and division are performed component-wise. In other words, for

addition, the i components are added together, the j components are added together, and the k components are added together. We also overloaded useful combinations of VEC3 objects and scalars. For example, the addition of a VEC3 object and a scalar is defined such that the scalar is added to each component of the vector. Some combinations are meaningless. For example, dividing a scalar by a vector.

Another class called ROT was created to support rotational transformations. A ROT is a 3-by-3 matrix made of three 3-dimensional column vectors, each VEC3s. Each column can be individually accessed as vectors n (normal), o (orientation), and a (approach), which is consistent with standard homogeneous transformation terminology [9]. Furthermore, each component of a vector can be accessed individually. For example, if a ROT ttt was declared, ttt.n refers to column n of the matrix, and ttt.n.i refers to the ith component of the nth column.This gives the programmer a clean way of accessing the different pieces of a matrix.

Having defined these two classes we can write a program which implements the Jacobian algorithm mentioned above. This program generates the Jacobian matrix for a PUMA 560. However, it is easy to modify it for any robot manipulator by changing the value of the constant REVOLUTE.

```
#include <stream.h>
#include <math.h>
#include "3D_classes.h"

const REVOLUTE = 0x7e     // bit code for revolute/prismatic combination for
                          // the PUMA manipulator
main()
{
int i;
float theta[7];

/* Declaration of 6-by-6 Jacobian Matrix */

VEC3  gamma[7];           // gamma[0] is not used
VEC3  beta[7];            // beta[0] is not used

for(i = 1; i < 7; i++){
        gamma[i] = VEC3(0.0, 0.0, 0.0);     // clear Jacobian
        beta[i] = VEC3(0.0, 0.0, 0.0);
}

ROT U[8];                 // initialize rotational matrices with the
U[1] = ROT(PI/2, -PI/2);  // Denivit-Hartenberg joint angle and twist angle
U[2] = ROT(PI/8, 0.0);
U[3] = ROT(PI/7, PI/2);
U[4] = ROT(PI/10, PI/2);
U[5] = ROT(PI/30, PI/2);
U[6] = ROT(PI/20, 0.0);
U[7] = ROT(0.0, 0.0);

VEC3 p[7];                        //initialize translation vectors
p[1] = VEC3(0.0, 0.0, 0.0);       // these vectors point from origin of one
p[2] = VEC3(431.8, 0.0, 149.9);   // coordinate frame to the next
p[3] = VEC3(-20.32, 0.0, 0.0);
```

```
p[4] = VEC3(0.0, -433.07, 0.0);
p[5] = VEC3(0.0, 0.0, 0.0);
p[6] = VEC3(0.0, 0.0, 56.25);
p[7] = VEC3(0.0, 0.0, 0.0);

ROT N_U[8];                          // initialize base to joint rotational matrices
N_U[7] = ROT(0.0, 0.0);              // initialize end-effector rotational matrix to I

VEC3 r[8];                           // initialize base to joint vector
r[7] = VEC3( 0.0, 0.0, 0.0);         // clear vectors
r[6] = VEC3( 0.0, 0.0, 0.0);

for (i = 6; i >= 0; i--){

        theta[i] = RD_THETA;         // read in new theta value
        U[i].load_theta(theta[i]);   // input new theta into U matrix

        N_U[i] = N_U[i+1] * U[i+1].T();
        r[i] = r[i+1] - N_U[i+1] * p[i+1];
}

for (i = 6; i > 0; i--){

        if((0x1 << i) & REVOLUTE) {
                gamma[i] = N_U[i-1].a;          // gamma[i] <- vector a in N_U[i-1]
                beta[i] = cross(gamma[i], -r[i-1]);     // cross product
        }
        else {
                gamma[i] = VEC3(0.0, 0.0, 0.0);
                beta[i] = N_U[i-1].a;
        }
        cout << beta[i] << gamma[i] << "\n";   // output ith column of the Jacobian
}
}
```

This program demonstrates how easily the Jacobian algorithm is written in C++ with the addition of the VEC3 and ROT classes. Furthermore, because the compiler is made aware of these new data types, the final code can exploit the vector nature of the robotic processor.

The arithmetic operators are overloaded to operate on the new data types. Consequently, arithmetic operators can be used instead of function calls, which simplifies the appearance and readability of the program. A couple of specialized functions are also defined. For example, U[i+1].T refers to the transpose of U[i+1], and U[i].load_theta(theta[i]) inserts the new theta value into the appropriate places of the rotational matrix.

C++ supports a mechanism called constructors [6], which are used to initialize user defined classes. VEC3 objects are initialized with the values of the three vector components. ROT data types are initialized with the Denivit-Hartenberg joint and twist angles [9]. Code in 3D_classes.h describes how to generate the 3-by-3 matrix from this information.

5. Conclusion

We are using the RIPS prototype as a test bed for experimenting with two-arm cooperative control. This experiment provides insight to two-arm control research, as well as tests the RIPS architecture. Much of our work has been the development of a programming environment which gives us the tools to perform our experiments.

We have written a monitor program, with a menu interface, for communicating with our custom hardware. A custom interface card which connnects the RIPS system to the two manipulators has been built. This card generates both high-accuracy position and velocity feedback from optical encoder signals.

High-level language support becomes necessary to implement complex control algorithms. We are currently targetting the GNU C++ optimizing compiler to the robotic processor architecture. Our approach will enable us to write programs in high-level constructs and still generate efficient robotic processor vector code.

Acknowledgements

We gratefully acknowledge the contributions of research assistant Vikram Koka for his assistance in software development. We would also like to thank Texas Instruments, Advanced Micro Devices, and Cypress Semiconductor for their generous donations of key components, and the Free Software Foundation for providing the basis for our retargetable optimizing compilers. This work has been funded by the National Science Foundation under grant number 0842415 and by the Allen-Bradley Division of Rockwell International with matching support of the State of California MICRO program.

References

[1] Butner, S., Wang Y., Mangasar, A., Jordan, S., "Design and Simulation of a Robotic Instruction Processing System," *Proc. of the IEEE Conf. on Robotics and Automation*, Phil., PA., April 1988.

[2] Wang, Yulun and Steven E. Butner, "RIPS: A Platform for Experimental Real-Time Sensory-based Robot Control" to appear in the *Trans. on Sys, Man, and Cybernetics*, 1989.

[3] Wang, Yulun, *RIPS: A Computer Architecture for Advanced Robot Control*, Ph.D. dissertation, University of California, Santa Barbara, May 1988.

[4] Asada, H., and T. Kanada," Design of Direct-Drive Mechanical Arms," *ASME Journal of Vibration, Acoustics, Stress, and Reliability in Design*, Vol 105, No 3., pp 312-316, 1983.

[5] Nakamura, Y., Nagai, K. and Y. Tsuneo, "Dynamic Stability in Coordination of Multiple Robotic Mechanisms," *Int. Journal on Robotics Research*, vol 8., No 2., 1989.

[6] Stroustrup, Bjarne,*The C++ Programming Language*, Addison-Wesley, Reading, Mass., 1986.

[7] Stallman, Richard, *Internals of GNU CC*, Free Software Foundation Inc., 1988.

[8] Orin, David E. and William W. Schrader, "Efficient Computation of the Jacobian for Robot Manipulators," *Int. Journal of Robotics Research*, vol. 3, No. 4, Winter 1984.

[9] Paul, R.P., *Robot Manipulators*, MIT Press, Cambridge, Mass., 1982.

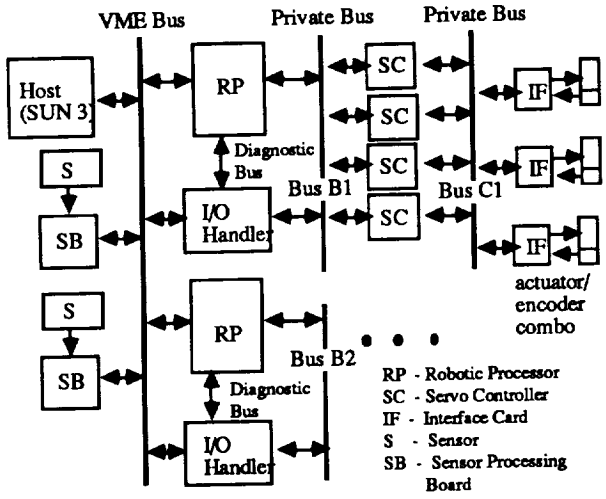

Figure 1. RIPS System Configuration

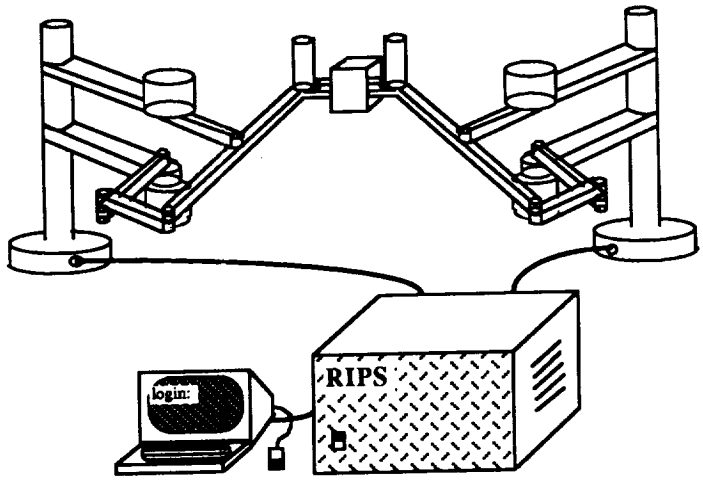

Figure 2. Two-Arm Set up

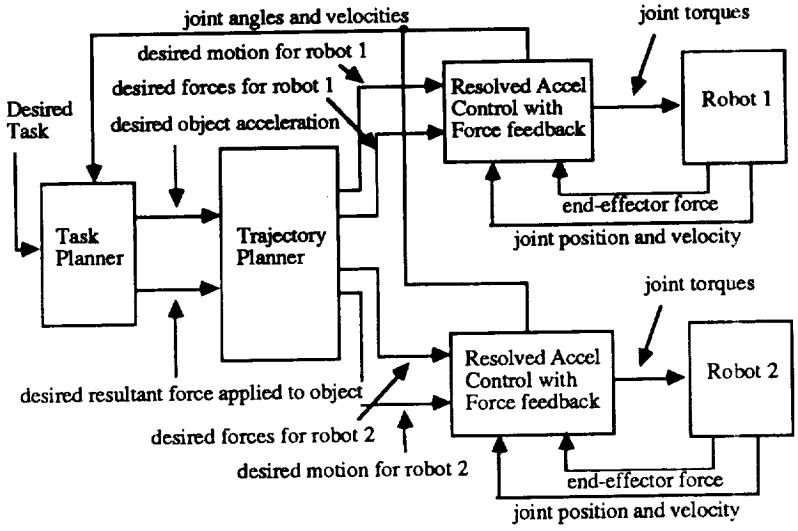

Figure 3. Dynamic Force Control of Two Arms

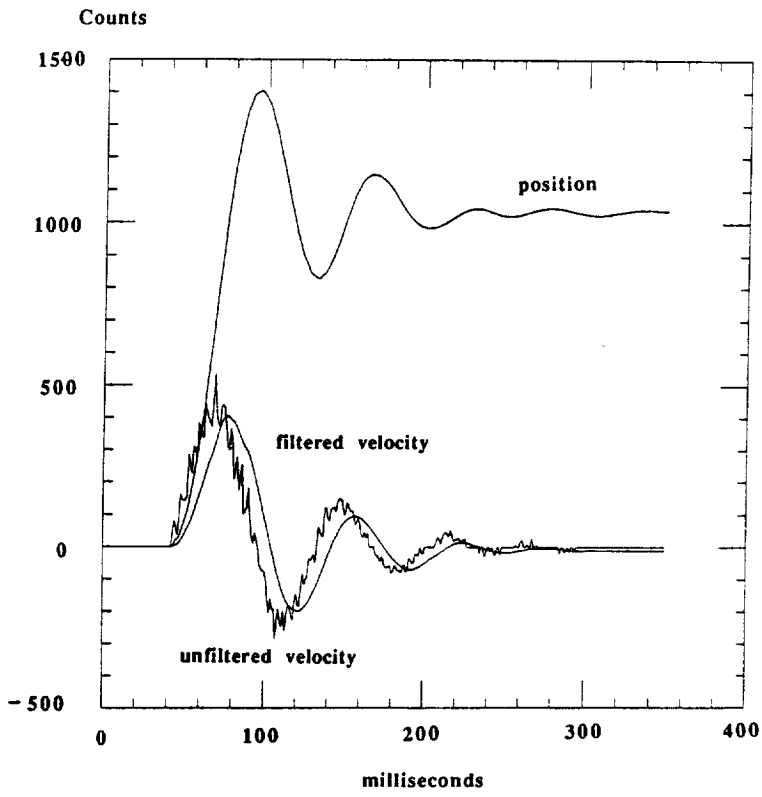

Figure 4. Measured Position and Velocity

```
rips:amante
                    * * * *  RIPS Command Menu  * * * *
RIPS: 1   RP0: 1    IOH: 1   SC: 1-----              Tue Oct 18 18:46:49 1988
                                                  |
0: quit:          Terminate session               |  10: rpcurd:    Read RPCU-Host
1: ripsreset:     Reset RIPS                      |  11: rpcurd3:   Read RPCU-Host vectors
2: ripstartup:    Test/start RIPS (NA)            |  12: rpcuwr:    Write RPCU-Host
3: ripstest:      Run Diagnostics (NA)            |  13: rpcuBrd:   Read RPCU-BusB
4: ripsrun:       Start RIPS                      |  14: rpcuBwr:   Write RPCU-BusB
5: ripshalt:      Halt RIPS                       |  15: rppmdown:  Download *.lst to RPPM
6: rpmenu:        Issue RP commands               |  16: rppmdump:  Dump RP program memory
7: scmenu:        Issue SC commands               |  17: rpdmdown:  Download data to RPDM
8: iohmenu:       Issue IOH commands              |  18: rpdmdump:  Dump RP data memory
9: testmenu:      RIPS tests                      |  19: rpcrrd:    Read RPCR
                                                  |  20: rpdmck:    Check RPDM
                                                  |  21: rppmck:    Check RPPM

  Enter command followed by <RETURN>: 19
  SWITCH = 0  RUN/PROG* = 1  HCROR= 0  HCRIR* = 0  HCUIR = 1  HCUOR = 0
  DATA = 0ff
```

Figure 5. Menu-Driven Monitor